An Introduction to R

Analytics Boot Camp
April 7, 2017

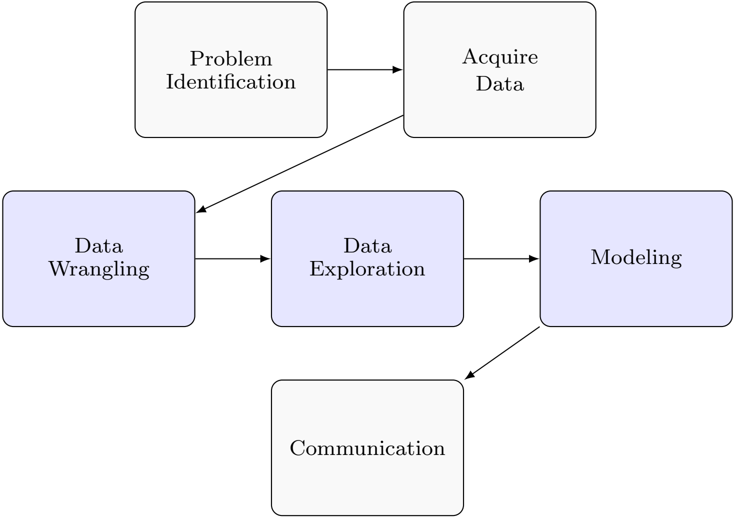
R

R is an open-source programming language for statistical computing, analytics, and data science.

R
cran.r-project.org

RStudio
www.rstudio.com

Analytics Process



Getting Familiar with R/RStudio

- Console: The best calculator *ever*.
- Source: Where you program (and save) your code.
- Environment: A look at what objects you have saved.
- Help: Your new best friend.

```
?c
```

- Cheatsheets.

Objects

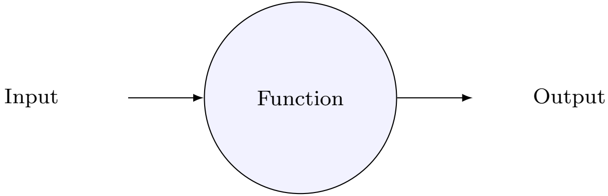
- R is an object-oriented programming language.
- What are objects?
  - Variables
  - Data
  - Functions
- The assignment operator `<-` or `=` assigns value to an object.

  ```
  x <- 2 * ( 3 + 1 )
  ```

- Object names are case-sensitive and can't include spaces.

  ```
  X
  jan rev <- 42000
  ```

Functions



- Functions are composed of arguments, ways to tweak how the function operates.
  - Don't forget you have help.
  - Make use of tab completion.
- Using a function is referred to as a "call" or a "function call."
- When possible, it is good coding convention to name non-input arguments in your function call.

Vectors

- A vector is a single column of data.

```
xNum  <- c(1, 3.14159, 5, 7)
xInt <- 1:4
xLog  <- c(TRUE, FALSE, TRUE, TRUE)
xChar <- c("yes", "no", "no", "yes")
```

- What happens when we mix types?

```
xMix  <- c(1, TRUE, 3, "Hello, world!")
```

- The elements in a vector can be vectors.

Summary, Length, and Structure

- The `summary()` function is a generic function that summarizes an object in a way that is (usually) appropriate for the type of data.

```
summary(xNum)
summary(xChar)
```

- Keep tabs on the dimensions of a vector using `length()` and (more generally) `str()`.

```
length(xInt)
str(xInt)
```

Indexing

- Indexing (subsetting, selecting) is about picking part of an object.

```
xNum
xNum[1]
xNum[2:4]
xNum[c(1,3)]
```

- Variables and math operators can be used as well.

```
start <- 2
xNum[start:(start+2)]
```

- A negative index omits elements.

```
xNum[-2]
```

Missing Values

- R treats missing values in a particular way.

```
soda_prices <- c(1.35, 1.50, 2, 1.75, 3.15, NA, NA)
```

- What's the average price?
- Never use another an actual numeric value for missing data.

Factors

- Nominal variables are stored as factors.

```
gender <- c(rep("male",20), rep("female", 30))
gender <- factor(gender)
```

- Ordinal variables are stored as ordered factors.

```
ranking <- c(rep("L",10),rep("M", 5),rep("S",20))
ranking <- factor(ranking,levels=c("S","M","L"),
                  ordered=TRUE)
```

Plot

- The `plot()` function is another generic function that handles an object in a way that is (usually) appropriate for the type of data.

```
plot(gender)
plot(ranking)
plot(soda_prices)
```

Other Useful Operators

- Operators that are frequently used, beyond the assignment operator, include:

```
2 == 2 # compare objects
c(1, 2, 3) == 2
```

```
1 != 2 # not equal to
```

```
2 < 2 # less than
```

```
2 <= 2 # less than or equal to
```

```
(2 < 2) | (2 <= 2) # or
```

```
(2 < 2) & (2 <= 2) # and
```

Good Coding Conventions

- Always comment your code.
- Use spaces to help your code be more readable.

  ```
  x<-1+(2*3)
  ```

  ```
  x <- 1 + ( 2 * 3 )
  ```

- Name objects clearly.

  ```
  janRev
  ```

  ```
  jan_rev
  ```

  ```
  jan.rev
  ```

- Structure your code linearly.
- Be consistent.

Beyond One Dimension

- Matrices: The two-dimensional extension of vectors.
- Arrays: Matrices with more than two dimensions.
- Data Frames: Like a matrix, except the columns can be of different data types.
- Lists: Like a vector, except each element can be of different data types and be a vector, matrix, list of lists, etc.

Data Frames

- Data frames are the most common way to handle data sets in R.

```
x.df <- data.frame(xNum, xLog, xChar)
```

- Like matrices, data frames are indexed by row, column.

```
x.df[2,1]
```

RC Cola Will Cure You

Indexing Data Fames

- Indexing along two dimensions is a natural extension of indexing vectors.

```
x.df[2,]
x.df[,3]
x.df[2:3,]
x.df[,-2]
```

Indexing by Name

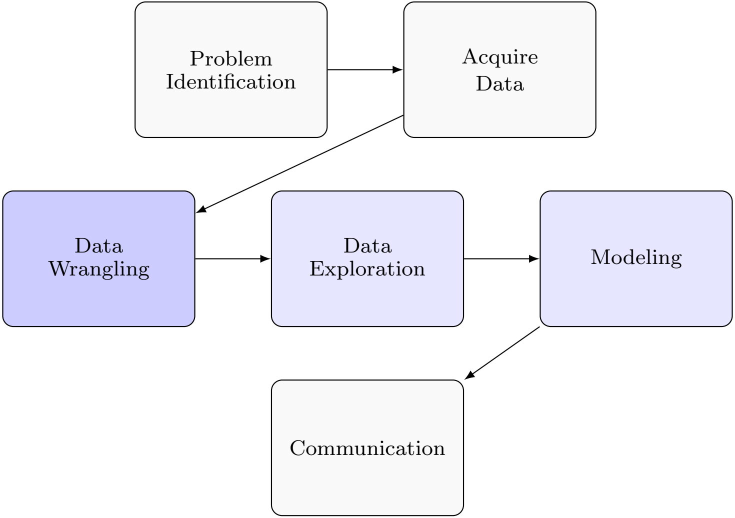
- An added benefit of data frames and lists is that the columns and list elements, respectively, can be named.

  ```
  names(x.df)
  ```

- You can use the numeric value or the associated names to index the columns or list elements.

  ```
  x.df$xLog
  x.df[,"xLog"]
  x.df[,3]
  ```

Analytics Process



The Working Directory

- The working directory is the directory that R has “open.”
- You want to set your working directory where you want to import data or export results.

```
getwd()              # Identify your working directory.
setwd("file path") # Set your working directory.
```

- There are significant differences in operating system file path syntax, so use RStudio to set your working directory.
- Copy and paste that code in your R script to quickly automate setting your working directory.

Importing Data

- A wrapper is a function wrapped around another function.

```
read.table("Sales.txt",header=TRUE,sep="\t")
sales <- read.delim("Sales.txt")
sales <- read.csv("Sales.csv")
```

- Note that loading data will overwrite same-named objects.

Exporting Data

- With an R script you can save code, but what about data?
- If we want to save data we need to export it.

```
write.table(sales)
write.table(sales,file="test.txt",row.names=FALSE)

write.csv(sales,file="test.csv",row.names=FALSE)
```

- The files are exported to your working directory.
- If you'll be importing the data back into R, export as raw data.

```
save(sales,file="test.RData")
rm(sales)
load("test.RData")
```

- But can't you just use RStudio's shortcuts?

Packages

- A package is collection of functions, documentation, and sometimes data.
- There are a number of packages that are part of the base installation of R.
- You can download other packages from CRAN and load them from their library directory.

  ```
  install.packages("tidyr")
  library(tidyr)
  ```

- Not all packages are created equal.

The tidyverse

- “The packages in the tidyverse share a common philosophy of data and R programming, and are designed to work together naturally.” –*R for Data Science*
  - readr – importing data
  - tidyr – cleaning data
  - dplyr – manipulating data
  - ggplot2 – visualize data
- The tidyverse is its own ecosystem of packages.

```
install.packages("tidyverse")
library(tidyverse)
```

- For a more detailed summary of the tidyverse, watch Hadley Wickham explain.

Inspect

- Inspecting the data (directly or graphically) is a great way to check for anything we need to wrangle.

```
summary(sales)
str(sales)
class(sales)
dim(sales)
names(sales)
head(sales)
tail(sales)
```

- Can we know what to look for if we don't know what the data is?

Tidy Data

1. Each column is a variable.
2. Each row is an observation.
3. Each value is in its own cell.
4. Each table has only one type of observational unit.

Gather and Spread Columns

- When you have columns that are really values, `gather()` columns into key-value pairs.

  ```
  gather(data, key, value, ...)

  head(sales)
  sales_long <- gather(sales,key=week,value=units,
                       week1:week7)
  head(sales_long)
  ```

- When you have values that should be columns, `spread()` key-value pairs into columns.

  ```
  spread(data, key, value)

  sales_wide <- spread(sales_long,key=week,value=units)
  head(sales_wide)
  ```

Separate and Unite Columns

- When you have two values in one column, `separate()` the values into two columns.

  ```
  separate(data, col, into, sep)

  sales_sep <- separate(sales,col=year_mo,into=c("year","mo
  head(sales_sep)
  ```

- When you have two values that should be in one column, `unite()` the values into one column.

  ```
  unite(data, col, ...)

  sales_uni <- unite(sales_sep,col=year_mo,year,month)
  head(sales_uni)
  ```

Selecting Variables

- Sometimes you only care about keeping certain variables.
- This is especially important with large datasets.

```
sales_1 <- select(sales, ID, week1:week2)
```

- To avoid errors, check your work as you go along.

```
head(sales_1)
```

Creating New Variables and Recoding

- We can create new variables using existing variables.

```
sales_2 <- mutate(sales_1, gt_5 = week1+week2 >= 5)
```

- We can recode an existing variable for clarity or to correct errors.

```
sales_2 <- mutate(sales_2, gt_5 = as.integer(gt_5))
```

Fusing Datasets

- Data fusion is about fusing datasets together.
- In the simplest case, a common variable (like an ID) allows us to merge the two data tables.

```
sales_3 <- left_join(sales, sales_2, by="ID")
```

Selecting Observations

- We often want to subset our data by keeping certain observations.

```
sales_4 <- filter(sales, week1 > 2)
```

Sorting Observations

- Sorting observations can reveal helpful information, provide a way to check data, and may be helpful for visualizations.

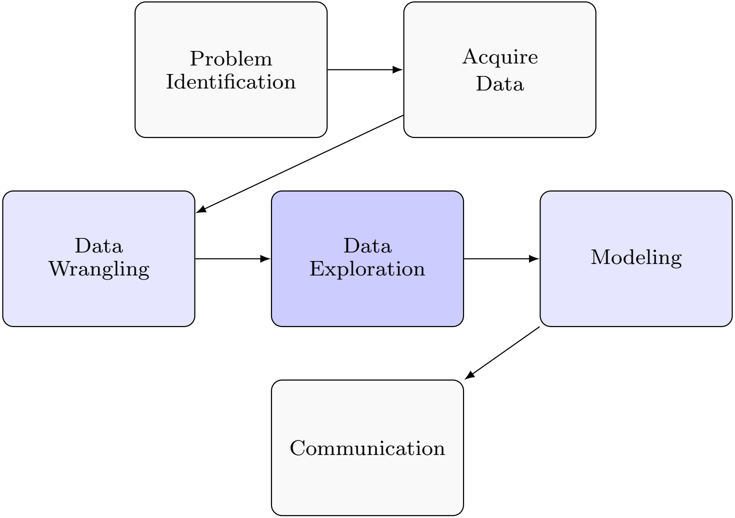
```
arrange(sales_4, desc(week1))
```

Pipes

- The pipe operator from the magrittr package (loaded along with dplyr) can be a helpful tool for writing clear code.

```
sales %>%
  filter(week1 > 2) %>%
  arrange(desc(week1))
```

Analytics Process



The Grammar of Graphics

- First, let's import new data, this time from the web!

```
store_df <- read.csv("http://goo.gl/QPDdMl")
str(store_df)
```

- ggplot2 is built using a consistent "grammar of graphics."
  - Data
  - Aesthetics – Mapping graphical elements to data.
  - Geometries – Graphic representing the data.

```
ggplot(store_df, aes(country)) +
  geom_bar()

table(store_df$country) %>%
  prop.table()
```

Comparing Variables

- When we want to start comparing variables, ggplot2 really begins to shine.

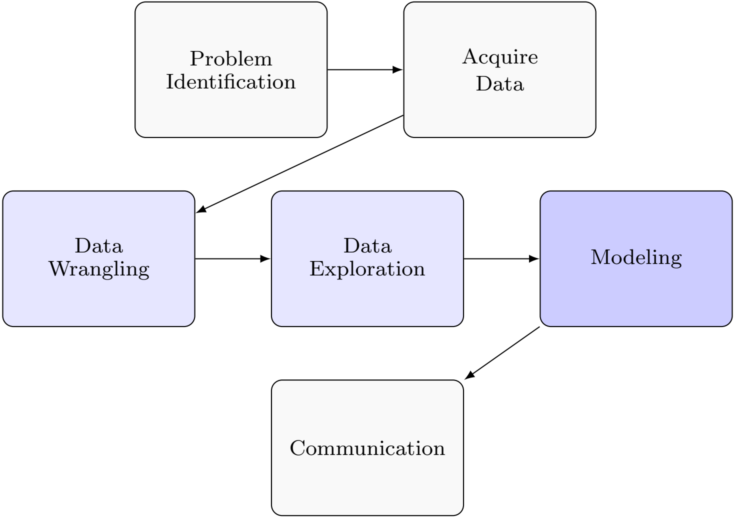
```
store_df %>%
  mutate(gt_150=p1sales > 150) %>%
  ggplot(aes(x=country,fill=gt_150)) +
    geom_bar()

store_df %>%
  mutate(p2prom=as.factor(p2prom)) %>%
  ggplot(aes(x=p2sales, fill=p2prom)) +
    geom_density(alpha=0.5)
```

More Advanced Graphing Options

- The structure of the data will determine what can be plotted.

```
store_df %>%
  mutate(p1prom=as.factor(p1prom)) %>%
  ggplot(aes(x=p1sales,y=p1price,col=p1prom)) +
    geom_jitter(alpha=0.5,size=2) +
    geom_smooth(method="lm",se=FALSE) +
    facet_wrap(~ country)
```

Analytics Process



Regression

- Models (e.g., hypothesis tests and regressions) are needed to say something about a population of interest.
- In regression, we usually specify a linear relationship.

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p + \varepsilon$$

- Regression models are used for both description and inference.
  - Is y related to x?
  - How are they related?
  - What is the strength of the relationship?
- They can also be used to predict unknown values of y.

Fit a Multiple Regression Model

- We want to know which features of the park are related to overall satisfaction.
- To do that we fit a model where all the features of the park are included as predictors.

```
out <- lm(p2sales ~ p2prom + p2price, data=store_df)
summary(out)

new_store_df <- data.frame(p2prom=c(0,0,1,1),
                           p2price=c(2.99,3.20))
predict(out,new_store_df)
```

Text Analysis

- R can work with unstructured data as well, like text.

```
install.packages("tm")
library(tm)
sylvan_data <- read.csv(file="sylvan_data.csv",
                        stringsAsFactors=FALSE)
comments_source <- VectorSource(sylvan_data$Comments)
comments_corpus <- Corpus(comments_source)

# WINDOWS: Remove non-UTF-8 characters.
comments_corpus <- tm_map(comments_corpus,
                content_transformer(function(x)
                iconv(enc2utf8(x), sub = "byte")))
# MAC: Remove non-UTF-8 characters.
comments_corpus <- tm_map(comments_corpus,
                content_transformer(function(x)
                iconv(x, to='UTF-8-MAC', sub = "byte")))
```

Functional Programming

- You can simplify your workflow by creating your own functions.

```
preprocess_corpus <- function(corpus) {
  # Make corpus lowercase; remove stopwords, punctuation,
  # and numbers; stem words; and strip whitespace.
  corpus <- tm_map(corpus,content_transformer(tolower))
  corpus <- tm_map(corpus,removeWords,
                   c(stopwords("english"),"sylvan"))
  corpus <- tm_map(corpus,removePunctuation)
  corpus <- tm_map(corpus,removeNumbers)
  corpus <- tm_map(corpus,stemDocument)
  corpus <- tm_map(corpus,stripWhitespace)
  return(corpus)
}
```

Create a Wordcloud

- User-generated packages cover an incredible breadth of applications.

```
install.packages("wordcloud")
library(wordcloud)

comments_corpus <- preprocess_corpus(comments_corpus)
wordcloud(comments_corpus,min.freq=1000,
          colors=c("grey","darkblue"))
```

Other Techniques

- Generalized linear models.
- Clustering and classification algorithms.
- Spatial and time series analysis.
- Sentiment analysis and topic modeling.
- Factor analysis and perceptual mapping.
- Interface with SQL.
- Create analytics dashboards.

Cleaning Up

- As a general rule, don't save your workspace when quitting R but do save changes to any R scripts that you wish to keep.
- A lot of mistakes happen because you don't have a clean workspace – so keep it clean.

Why Use R?

- It's free and open source.
- Widest range of established and emerging techniques.
- New analytic techniques are often first introduced in R.
- Known for being able to produce world-class graphics.
- R syntax is incredibly flexible and fairly easy to learn.
- Frequently used tasks can be easily automated.
- A very active and helpful community.
- R skills are in high demand.
- Everybody's using it.

Where Can I Learn More?

- Data Camp
- R for Data Science
- Text Mining with R
- R for Marketing Research and Analytics